

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Venus Framework Saint Ouen |  | PAQUET Judicaël |  | Judicael.paquet@gmail.com  https://github.com/las93 |

Venus Framework 3 PHP

[Le nouveau framework MVC PHP]

Table des matières

1. [Présentation 2](#_Toc388455867)

[Installation](#_Toc388455868)

[Concept du routeur](#_Toc388455869)

[Routes avec des paramètres](#_Toc388455870)

[Routes complexes](#_Toc388455871)

1. [Nouvelle application 5](#_Toc388455872)

[Créer un nouveau projet](#_Toc388455873)

[Créer un contrôleur avancé](#_Toc388455874)

[Les modèles et entités de bases](#_Toc388455875)

[Les templates (vues) de bases](#_Toc388455876)

1. [Concepts avancés 9](#_Toc388455877)

[Utiliser un modèle différent du controller](#_Toc388455878)

[Des raccourcis dans les controller](#_Toc388455879)

[Manipulation des modèles](#_Toc388455880)

[Créer des entités/modèles en scaffolding](#_Toc388455881)

[Créer des vues](#_Toc388455882)

1. [Vos besoins 17](#_Toc388455883)

[Créer une page d’erreur 404/403](#_Toc388455884)

[Variables pré-définies et commentaires des templates](#_Toc388455885)

[Les « Modifiers » des templates](#_Toc388455886)

[Les variables dans les templates](#_Toc388455887)

[Les boucles et conditions dans les templates](#_Toc388455888)

[Les fonctions dans les templates](#_Toc388455889)

[Aouter des méthodes avec l’ORM dans mon modèle](#_Toc388455890)

# Présentation

Venus Framework est une nouvelle alternative de qualité de développement open-source dans les environnements LAMP fonctionnant sous un système MVC complet.

## Installation

L’installation du Framework est relativement simple. Vous pouvez télécharger la version .zip sur le Github <https://github.com/las93/venus3>

Pour installer Venus Framework 3, il faut au préalable avoir un PHP 7 d’installé.

Le framework de base permet de lancer un serveur web afin de tester vos codes en allant dans le dossier racine du framework et en faisant :

|  |
| --- |
| **php console/bin server:run** |

\* Cette syntaxe a été inspire de Symfony afin que les nouveaux utilisateurs ne soient pas perdus

Vous pouvez également installer Apache (Nginx/IIS) pour faire office de server web.

Venus Framework 3 ne fonctionne qu’avec PHP 7 et n’est compatible avec aucune version de PHP 5. Nous recommandons d’installer Mysql si vous avez besoin d’une base de données pour vos projets.

De base, il existe un projet Demo pour vérifier votre installation. Il faut ajouter un vhost dans vos configurations Apache pour installer Venus Framework.

<VirtualHost \*:80>  
 ServerName localhost  
 DocumentRoot E:/venus/bundle/src/Demo/public/

<Directory E:/venus/bundle/src/Demo/public/>  
 DirectoryIndex index.php  
 AllowOverride All  
 Order allow,deny  
 Allow from all  
 Require all granted  
 </Directory>  
</VirtualHost>

Vous pouvez insérer les .htaccess dans vos configurations apache pour des raisons de performances.

## Concept du routeur

Le .htaccess est la base du routeur de notre framework. Il va router toutes les demandes vers le bootstrap (fichier de lancement) index.php du framework. Cependant, si un des fichiers demandés existent (comme une image par exemple), il retournera le contenu du fichier. Nous vous recommandons d’ailleurs de ne pas créer un autre fichier dynamique (type PHP) et de toujours travailler au sein du framework.

RewriteEngine on  
RewriteCond %{REQUEST\_FILENAME} !-f  
RewriteRule ^.\*$ /index.php [NC,L]

Le bootstrap lance le routeur du framework qui s’occupera d’appeler les méthodes nécessaires pour le bon fonctionnement de votre application.

Le routeur ne fait que respecter ce que vous lui définissez dans les fichiers de configuration. De base, le framework vous propose une configuration pour notre projet Demo :

{

"localhost" : {

"routes": {

"home": {

"route": "/",

"controller" : "\\Venus\\src\\Demo\\Controller\\Home",

"action": "show"

}

}

}

}

Les fichiers de configurations sont des fichiers Json. Vous pouvez d’ailleurs les faire valider sur ce site lors de vos modifications car l’erreur est très rapide :  
http://jsonlint.com/

« Localhost » est le domaine de votre application (ou la base). Vous pouvez également faire « localhost/demo ».

Dans notre fichier, nous définissons ensuite nos routes.

« home » est un simple alias qui sera cependant important car il permettra de créer des liens dynamiquement sur vos applications.   
« route » permet de définir l’URL a appeler pour lancer une action. Nous verrons plus tard, qu’il est possible de créer des URL dynamique.  
« controller » permet de définir la classe qui contiendra l’action à enclencher (le nommage respecte les namespace du framework. Même si cette méthode est assez ouverte, il est recommandé de suivre les normes de nommage comme vous le voyez dans l’exemple.  
« action » permet de définir l’action (en gros la méthode de la classe) à déclencher.

Vous pouvez aller voir le fichier /Venus/bundles/src/Demo/app/Controller/Home.php et la méthode show() pour voir comment le framework affiche un simple Hello World.

## Routes avec des paramètres

Les routes peuvent être plus dynamiques et prendre en compte par exemple des paramètres :

{

"localhost" : {

"routes": {

"home": {

"route": "/[:id]",

"controller" : "\\Venus\\src\\Demo\\Controller\\Home",

"action": "show",  
 "constraints": {

"id": "[0-9]\*"

}

}

}

}

}

On a rajouté ici un paramètre id dans la « route » que l’on définit plus précisément dans les « constraints » sous forme d’expression régulière. Il faudra également rajouter $id dans la méthode show(). Les paramètres sont également automatiquement rajoutés dans le $\_GET.

Pour gérer une configuration alternative (pour un environnement de développement par exemple), vous pouvez créer un fichier Route.conf-local qui d’ailleurs est prioritaire sur le fichier Route.conf.

Il est possible de définir des GET par défaut car cela peut s’avérer très utile. Voici un exemple utilisé dans Hélium :

"a-propos-d-helium": {

"route": "/a-propos-d-helium.html",

"controller" : "\\Venus\\src\\Helium\\Controller\\FreeContent",

"action": "index",

"defaults\_constraints": {

"sPage": "a-propos-d-helium"

},

"constraints": {

"sPage": "a-propos-d-helium"

}

},

## Routes complexes

Venus Framework est un framework MVC donc il propose un système de templating complexe. Cependant pour des pages statiques, il permet de diriger une route directement vers un template pour omettre les phases MC.

Voici comment écrire par exemple sa route :

"template": {

"route": "/template/",

"template": "Demo",

"layout": false,

"content\_type": "html"

}

Nous allons créer également un fichier e:\venus\bundles\src\Demo\app\View\Demo.tpl avec du texte quelconque dedans. Le « template » comprend automatiquement le lieu où chercher le template ainsi que l’extension .tpl de celui-ci.

Le « layout » permet d’automatiser la présence d’un layout. Pour ajouter ce Layout, il suffit de créer un fichier e:\venus\bundles\src\Demo\app\View\Layout.tpl et d’ajouter cela à l’intérieur pour appeler le modèle définit : {include file=$model}

Vous pouvez ainsi tester de mettre true à « layout ».

Dans une route il est possible également de préciser la méthode http utilisée : GET, POST, HEAD, PUT, DELETE, TRACE, CONNECT pour créer des services full-rest par exemple.

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"methods": "GET"

},

Vous pouvez également mettre du cache sur la page qui permettra d’indiquer le temps de vie de celle-ci.

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"cache\_max\_age": 100

},

Pour mettre en place des pages en https, vous pouvez également indiquer que la page sera en HTTPS :

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"schemes": "https"

},

# Nouvelle application

## Créer un nouveau projet

Pour se lancer dans l’aventure, le mieux c’est de se créer un nouveau projet. Pour cela, il faut directement travailler en ligne de commande afin de lancer le script de création de projet.

|  |
| --- |
| **/!\ Ne pas copier/coller ce code, word transforme certains caractères que Linux ne reconnait pas.** |

Sous Linux : />php bin/console create\_project –p Front

Sous Windows : C:\Users\judicael.paquet>php bin/console create\_project -p Front

Votre projet (qu’on appelle bundle depuis Venus Framework 2) Front/ est à présent créé avec un espace bundles/src/Front/public et un espace applicatif bundles/src/Front/app.

## Créer un contrôleur avancé

Pour le moment nous avons vu qu’un contrôleur simple qui affichait un Hello World.

Comme dans de nombreux framework, le routeur va gérer la vue associée et le modèle (sous forme d’ORM complexe et d’entité). Nous allons commencer par regarder notre fichier e:/venus/bundles/src/Demo/app/Controller/Exemple1.php qui contient un exemple d’appel de template.

|  |
| --- |
| class Exemple1 extends Controller {  /\*\*  \* the main page  \*  \* @access public  \* @return void  \*/  public function show() {  $aExemple = $this->model->get();    $this->view  ->assign('Exemple', $aExemple)  ->display();  } ) |

Pour faire simple, la fonction appelle un modèle Exemple1 ($this->model est raccourcis vers le modèle du même nom que le contrôleur) et ensuite on appelle la vue qui affiche la liste des Exemple1 qu’on a récupéré. Comme pour le modèle, si on ne définit pas la vue à appeler, le framework va automatiquement appeler la vue du même nom soit ici Exemple1.tpl.

Pour tester en réel, ce code, il nous faut se connecter sur la base de donnée et de créer la table Exemple1 comme ceci :

CREATE DATABASE demo

Puis :

CREATE TABLE `Exemple1` (  
  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,  
  `title` varchar(150) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1

INSERT INTO `demo`.`Exemple1` (`id` ,`title`) VALUES (NULL , 'test1'), (NULL , 'test2');

Le système propose du scaffolding mais nous verrons cela plus tard.

Dès que cela est fait, il faut aller configurer la connexion de sa base de données dans le fichier e:/venus/bundles/src/Demo/app/conf/Db.conf pour mettre l’utilisateur et le passeport de la base de données.

{

"configuration": {

"demo": {

"type": "mysql",

"host": "localhost",

"db": "demo",

"user": "root",

"password": "",

"tables": {

"Exemple1": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

}

}

}

}

\* Si vous changez de base, il faut impérativement modifier le bundles/conf/Const.php et le champ « db\_conf ».

Comme vous pouvez d’ailleurs le voir, ce fichier vous permettra de créer l’ensemble des outils automatiquement à l’avenir pour utiliser vos bases de données.

Ajouter ensuite dans votre fichier de route, la nouvelle route pour tester ce contrôleur :  
e:/venus/bundles/src/Demo/app/conf/Route.conf

"model": {

"route": "/model",

"controller" : "\\Venus\\src\\Demo\\Controller\\Exemple1",

"action": "show",

}

Si vous avez réussi vous pourrez avoir dans votre navigateur :



## Les modèles et entités de bases

Afin de machin l’ensemble du travail du développeur, les bases de données sont gérées de façon pseudo-automatique.

Notre table Exemple1 a un modèle et une entité au sein du framework. Une **entité** est un conteneur représentatif de la table (avec des possibilités d’extensions) et le **modèle** est le gestionnaire de cette entité avec de nombreuses fonctions de manipulations de bases (plus de 90% des besoins).

Le modèle est une classe vide dans laquelle on peut réaliser des demandes ORM plus complexes afin de répondre à l’ensemble des besoins de manipulations de Base de données. Voici un exemple :

|  |
| --- |
| /\*\*  \* Get Lasts folders  \*  \* @access public  \* @param integer $iLimit limit  \* @return array  \*/  public function getLastFolders($iLimit = 3) {  $aJoin = [  [  'type' => 'right',  'table' => 'article\_type',  'as' => 'at',  'left\_field' => 'at.id',  'right\_field' => 'a.id\_article\_type'  ]  ];  $result = $this->orm  ->select(array('SQL\_CALC\_FOUND\_ROWS', '\*'))  ->from($this->\_sTableName, 'a')  ->join($aJoin)  ->where(array('a.id\_article\_type' => 3))  ->orderBy(array('a.'.LibEntity::*getPrimaryKeyName*($this->entity).' DESC'))  ->limit($iLimit)  ->load();  $result[0]->count = $this->orm  ->select(array('FOUND\_ROWS()'))  ->load();  $result[0]->pages = *floor*($result[0]->count / 10);  if (*isset*($result)) { return $result; }  else { return array(); }  } |

Entité : e:/venus/bundles/src/Demo/app/Entity/Exemple1.php  
Modèle : e:/venus/bundles/src/Demo/app/Model/Exemple1.php

## Les templates (vues) de bases

Un template, outil efficace pour tout framework, a pour but de dissocier l’affichage et la gestion de donnée. Il propose cependant des options dynamiques afin de proposer un maximum de flexibilité tout en gardant une séparation propre.

Voici le template de notre exemple précédent : e:/venus/bundles/src/Demo/app/View/Exemple1.tpl

{foreach from=$Exemple key=$iKey item=$oExemple}

{$oExemple->get\_title()} - {$oExemple->get\_id()}<br/><br/>

{/foreach}

Notre template contient de l’HTML et un balisage particulier permettant de boucler sur les deux exemples que l’on a insérés en base.

Nous verrons plus tard toutes les options proposées par le système de templating de venus Framework.

# Concepts avancés

## Utiliser un modèle différent du controller

Nous avions vu pour faire une action qui appelait un modèle/entité du même nom mais vous aurez besoin d’appeler d’autres modèles/entités voire vous voudrez totalement dissocier le nom du controller aux noms des modèles/entités.

Pour cela nous allons créer une nouvelle table news dans notre base de données Mysql demo.

CREATE TABLE `news` (  
  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,  
  `title` varchar(150) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

INSERT INTO `demo`.`news` (`id` ,`title`) VALUES (NULL , 'news1'), (NULL , 'news2');

Nous rajoutons notre table ensuite dans le DB.conf :

"news": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

Nous créons ensuite l’entité correspondante dans e:/venus/bundles/src/Demo/app/Entity/news.php

|  |
| --- |
| <?php  /\*\*  \* Model news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  namespace Venus\src\Demo\Entity;  use \Venus\src\Demo\common\Entity as Entity;  /\*\*  \* Model news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  class news extends Entity {  /\*\*  \* id  \*  \* @access private  \* @var int  \*  \* @primary\_key  \*/  private $id = null;  /\*\*  \* title  \*  \* @access private  \* @var string  \*  \*/  private $title = null;  /\*\*  \* get id of article  \*  \* @access public  \* @return int  \*/  public function get\_id() {  return $this->id;  }  /\*\*  \* set id of article  \*  \* @access public  \* @param int $id id of article  \* @return \src\FrontOffice\Entity\article  \*/  public function set\_id($id) {  $this->id = $id;  return $this;  }  /\*\*  \* get title of article  \*  \* @access public  \* @return string  \*/  public function get\_title() {  return $this->title;  }  /\*\*  \* set title of article  \*  \* @access public  \* @param string $title title of article  \* @return \src\FrontOffice\Entity\article  \*/  public function set\_title($title) {  $this->title = $title;  return $this;  }  } |

Nous créons ensuite le modèle correspondant dans e:/venus/bundles/src/Demo/app/Model/news.php

|  |
| --- |
| <?php  /\*\*  \* Entity news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  namespace Venus\src\Demo\Model;  use \Venus\src\Demo\common\Model as Model;  /\*\*  \* Entity news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  class news extends Model {  } |

A présent nous allons ajouter l’utilisation de ce modèle dans notre controller afin de bien comprendre la rapidité et l’efficacité du framework (c’est notre méthode recommandée mais ce n’est pas la seule méthode possible). Nous allons dans notre controller e:/venus/bundles/src/Demo/app/Controller/Exemple1.php

Nous commençons par rajouter l’alias « news » de la classe modèle en haut du fichier :

|  |
| --- |
| namespace Venus\src\Demo\Controller;  use \Venus\src\Demo\common\Controller as Controller;  use \Venus\src\Demo\Model\news as news; |

Ensuite nous créons notre raccourci d’accès au modèle dans le constructeur comme ceci :

|  |
| --- |
| public function \_\_construct() {  $this->modelNews = function() { return new news; };    parent::*\_\_construct*();  } |

Ensuite il ne nous reste plus qu’à modifier notre action show pour appeler ce nouveau modèle :

|  |
| --- |
| public function show() {  $aExemple = $this->modelNews->get();  $this->view  ->assign('Exemple', $aExemple)  ->display();  } |

Ensuite il suffit de vérifier le résultat dans notre navigateur :



## Des raccourcis dans les controller

Les controllers proposent des raccourcis intéressants qui pourraient bien vous aider à accélérer vos développements. Voici la liste de ceux-ci et les exemples associés.

Redirect

Vous pouvez créer une redirection http rapide (ou redirection interne en cli) comme ceci :

$oUrlManager = **new \Venus**\core\UrlManager;

$this->redirect($oUrlManager->getUrl('home', **array**()));

Forward

Vous pouvez créer une redirection vers une autre action comme ceci :

$oUrlManager = **new \Venus**\core\UrlManager;

$this->forward($oUrlManager->getUrl('home', **array**()));

Attention : La redirection n’est pas au niveau http, donc il n’y a pas de redirection 301 mais juste que le programme se dirige vers une autre action.

Cache

Vous pouvez cacher le développement d’un contrôleur en ajoutant un élément dans le PHPDoc de la méthode comme ceci :

/\*\*

\* The main menu manager

\*

\* **@access** public

\* **@return** void

\*

\* @Cache(maxage=12)

\*/

**public function** show() {

Raccourcis

Voici des raccourcis automatiquement propose à tous les contrôleurs que nous verrons plus en détail par la suite :

$this->form = function() { return new Form(); };  
$this->security = function() { return new Security(); };  
$this->router = function() { return new Router; };  
$this->mail = function() { return new Mail; };  
$this->session = function() { return new Session; };  
$this->translator = function() { return new I18n; };

Vous pouvez automatiquement envoyer des erreurs HTTP dans les contrôleurs afin de gérer vous-mêmes des retours HTTP :

$this->NotFound();  
$this->Forbidden();

## Manipulation des modèles

Les modèles de Venus Framework sont simples à utiliser et complexes à la fois. Vous avez déjà vu précédement dans ce document un simple ->get() pour récupérer une liste de données sous forme de tableau d’objet.

|  |
| --- |
| public function show() {  $aExemple = $this->modelNews->get();  $this->view  ->assign('Exemple', $aExemple)  ->display();  } |

Le get() renvoi un tableau de ce type :

Array(new news(), new news());

Ceci permet de pouvoir gérer facilement l’ensemble des données :

foreach ($aExemple as $iKey => $oNews) {  
 echo $oNews->get\_title() . “<br/>”;  
}

Voici les fonctions natives des modèles :

|  |  |
| --- | --- |
| get() ou get($ocriteria) | Obtenir une liste |
| update(($ocriteria) | Modifier |
| insert(($ocriteria) | Insérer |
| delete(($ocriteria) | supprimer |
| truncate() | Vide la table complète |

Vous pouvez également utiliser des méthodes très utiles comme décrites ci-dessous :

$oModel->findOneByid(12); (quand on a un champ id) => retourne un seul résultat

$oModel->findByid(12); (quand on a un champ id) => retourne tous les résultats correspondant

$oModel->findAll();=> retourne tout

$oModel->findOneBy(array('id' => 12); => retourne un résultat mais on passe les critères sous forme de tableau (permet de definir plusieurs critères)

$oModel->findBy(array('id' => 12); => retourne les résultats mais on passe les critères sous forme de tableau (permet de definir plusieurs critères)

Il est possible également de modifier ou de supprimer directement une entité pour gagner du temps :

$oEntity->save() ; ou $oEntity->remove() ;

## Créer des entités/modèles en scaffolding

Le scaffolding est un concept de création de classes à partir d’une base de données. Même si notre scaffolding se base sur le Db.conf, il a pour but de créer les entités et modèles automatiquement afin de représenter les tables sous forme d’objets manipulables.

Nous allons créer une nouvelle table avec notre moteur de scaffolding en ajoutant un schéma dans notre DB.conf de notre projet e:/venus/bundles/src/Demo/app/conf/Db.conf :

"article": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

Il faut ensuite lancer le script pour le scaffolding :

**/!\ Ne pas copier/coller ce code, word transforme certains caractères que Linux ne reconnait pas.**

Sous Linux : />php bin/console scaffolding:run –p Demo -c

Sous Windows : C:\Users\judicael.paquet>php bin/console scaffolding -p Demo –c

La table va se créer en base de données et il va également créer l’entité et le modèle correspondant dans notre projet.

-c => permet de créer les tables en base de données (si elles n’existent pas)

-e => permet de créer les entités et les modèles

-d => permet de supprimer les tables si elle existent (ça vide aussi les tables)

-f => permet de créer les modèles que si le modèle n’existe pas

## Créer des vues

Dans un framework MVC, il est indispensable de gérer des vues. Sous venus Framework, les vues sont des templates qui proposent de nombreuses possibilités dynamiques.

Voici un exemple de vue pour mieux comprendre déjà inclus dans le package que vous avez téléchargé :

|  |
| --- |
| {foreach from=$Exemple key=$ikey item=$oExemple}  {$oExemple->get\_title()} - {$oExemple->get\_id()}<br/><br/>  {/foreach} |

Les templates proposent comme vous le voyez ci-dessus, un foreach proche du php pour parcourir un tableau complet.

Les templates proposent deux types de méthodes : les modifiers et les fonctions.

Les modifiers vont permettre de modifier la variable proposée juste avant de l’afficher comme l’exemple ci-dessous :

{$title|capitalize} => va mettre le contenu de $title avec une majuscule sur le premier caractère.

Les fonctions qui vont demander des paramètres d’entrées, vont permettre d’excuter quelque chose (et pas appliquer une simple modification avant sortie). Voici un exemple :

{foreach from=$array key=$iKey item=$oArticle}{/foreach} => comme un foreach en PHP.

Une vue n’a accès aux variables que si le contrôleur lui en donne l’autorisation en assignant une variable au template. Nous l’avions déjà vu dans ce document sous cette forme :

|  |
| --- |
| $this->view  ->assign('Exemple', $aExemple)  ->display(); |

Dans le contrôleur, on assigne une variable au template (assign() ) avant delancer le template (display() ). La variable $Exemple sera donc utilisable dans le template appelé.

# Vos besoins

## Créer une page d’erreur 404/403

Il est bien pratique de pouvoir créer des pages d’erreurs personnalisées pour les applications de type web. Venus Framework propose de mettre en place ce type de page très rapidement.

Dans le package de base, vous avez déjà un exemple en place (e:/venus/bundles/conf/Route.conf) où l’on ajoute une route puis le template. Voici la route à créer :

|  |
| --- |
| "routes": {  "404": {  "template": "Error",  "vars":{  "error\_message": "Not Found",  "error\_code": "404"  },  "content\_type": "html",  "cache": {  "max\_age": 100  },  "methods": "GET"  }  } |

Les routes 404 et 403 sont automatiquement reconnues comme les routes de pages d’erreur. Les vars permettent de passer automatiquement des variables au template appelé dans la conf (cela permet d’appeler le même template avec des variables différentes).

Il vous suffira de modifier e:/venus/bundles/src/Demo/app/conf/Route.conf et e:/venus/bundles/src/Demo/app/View/Error.tpl pour personnaliser vos pages d’erreurs.

## Variables pré-définies et commentaires des templates

Dans les templates du venus Framework, il existe de nombreuses variables pré-définies afin d’accélérer la tache des développeurs/intégrateurs.

{counter} => permet d’afficher le numéro de boucles (pour les for par exemple)

{$SCRIPT\_NAME} => permet d’afficher la variable $\_SERVER[‘SCRIPT\_NAME’]

{\* et \*} => entourent des commentaires. Ceci ne s’affichera pas sur le site, c’est uniquement pour mettre des repères pour les développeurs/intégrateurs au sein du code source.

{literal} et {/literal} => permettent d’entourer du contenu statique. En gros le moteur de templating n’ira pas retraduire ce qu’il lit entre ces deux balises. C’est essentiel pour faire du javascript où les caractère { et } portent confusion avec le framework.

{$app.server} => représente la variable PHP $\_SERVER ;

{$app.get} => représente la variable PHP $\_GET ;

{$app.post} => représente la variable PHP $\_POST ;

{$app.cookies} => représente la variable PHP $\_COOKIES ;

{$app.env} => représente la variable PHP $\_ENV ;

{$app.session} => représente la variable PHP $\_SESSION ;

{$app.request} => représente un merge de toutes les variables tableaux dans ce sens strictement prioritaire : $\_GET, $\_POST, $\_COOKIES, $\_SERVER, $\_ENV.

{$app.now} => représente le timestamp du moment

{$app.conf} => représente toutes les constantes PHP définient.

{$app.template} => affiche le nom du template en cours

{$app.template\_object} => affiche l’object du template en cours.

{$app.current\_dir} => affiche le dossier du template en cours

{$app.version} => représente le numéro de version du système de template utilisé

{$app.ldelim} et {$app.rdelim} => représente les délimiteurs définissant {literal} et {/literal} que l’on peut reconfigurer à sa guise.

## Les « Modifiers » des templates

Les modifiers permettent de modifier une variable avant de l’afficher. Voici les différents modifiers que vous pourrez utiliser dans vos templates.

{$test|capitalize}

permet de mettre chaque première lettre des mots en majuscule

{$test|capitalize:true}

Le premier paramètre permet de définir si on transforme ou non les mots qui ont un caractère numérique comme par exemple php5.

{$test|capitalize:true:true}

Le deuxième paramètre permet de définir si tous les autres caractères sont transformés en caractères minuscules.

{$test|cat:' ajouté du texte'}

permet de mettre chaque première lettre des mots en majuscule

{$test|count\_characters}

permet de compter le nombre de caractères sauf les espaces

{$test|count\_characters:true}

permet de compter le nombre de caractères avec les espaces

{$test|count\_paragraphs}

permet de compter le nombre de paragraphes

{$test|count\_sentences}

permet de compter le nombre de parties séparées par des ., ?, !.

{$test|date\_format}

Affiche une date sous la forme : Mon, 19 May 2014 17:42:51

{$test|date\_format:'%b %e, %Y'}  
Formater sa date (voir en php strftime). %h devient %b, %n devient \n, %r devient %I:%M:%S %p, %R devient %H:%M, %t devient \t et %T devient %H:%M:%S.

{$test|default:'texte par défaut'}

permet d’afficher un texte par défaut si la variable est vide ou null.

{$test|escape}

permet d’échapper selon différents critères désirés : html, htmlall, url, urlpathinfo, quotes, javascript.

{$test|escape:'url'}

{$test|escape:'html':'UTF-8'}

{$test|from\_charset}

permet d’encoder du charset ISO-8859-1 au charset actuel

{$test|from\_charset:' ISO-8859-1'}

Permet d’encoder du charset precisé au charset actuel

{$test|hour}

permet d’afficher que 17:12 quand on met 2014-05-05 17:12:30

{$test|indent}

permet d’indenter du texte (en HTML il faut mettre des balises <bloquote>) de 4 espaces

{$test|indent:4}

{$test|indent:4:' '}

Voici les paramètres si vous désirez les changer.

{$test|regex\_replace:'/oui/':'non'}

Remplacer les oui par non par expression régulière

{$test|replace:'oui':'non'}

Remplacer les oui par non comme un simple str\_replace

{$test|spacify}

Sépare chaque mot par un espace

{$test|spacify:'^^'}

Sépare chaque mot de ^^

{$test|string\_format:'%d'}

Transformer une chaîne ou un chiffre sous le format désiré (voir sprintf en PHP pour les formats)

{$test|strip}

Remplace tous les caractères d’espacement en simple espace

pour les formats)

{$test|strip:'&nbsp;'}

Remplace tous les caractères d’espacement en &nbsp;

{$test|strip\_tag}

Remplace toutes les balises par un espace

pour les formats)

{$test|strip:false}

Remplace toutes les balises par rien

{$test|nl2br}

permet de transformer les \n en <br/>

{$test|to\_charset}

permet d’encoder du charset actuel au charset ISO-8859-1

{$test|to\_charset:' ISO-8859-1'}

Permet d’encoder du charset actuel au charset precisé

{$test|truncate}   
permet de couper les phrases à X caractères  
{$test|truncate:30:'…':false:false}  
Les paramètres permettre successivement de choisir le nombre de caractère avant de couper le contenu, ensuite de décider ce qu’on ajoute après le contenu comme les 3 points, ensuite on détermine si on coupe après le dernier mot ou directement au caractère 30 et le dernier paramètre permet de définir si on coupe en prenant le milieu de la chaîne de caractère en référence et non le début de la chaine.

{$test|unescape}

permet d’échapper selon différents critères désirés : html, htmlall

{$test|unescape:'html'}

{$test|unescape:'html':'UTF-8'}

{$test|lower}

permet de mettre toutes les lettres en minuscule

{$test|upper}

permet de mettre toutes les lettres en majuscule

{$test|wordwrap}

permet d’aller à la ligne tous les 75 caractères.

{$test|wordwrap:30:'<br/>':true}

Permet de mettre des <br/> pour aller à la ligne en HTML tous les 30 caractères et au caractère exact (et non après le dernier mot en mettant false).

## Les variables dans les templates

Les variables du moteur de template de Venus Framework est relativement simple à utiliser tout en proposant une grande complexité.

Si vous assignez un tableau avec un index numérique, vous pourrez afficher le contenu du tableau.

|  |  |  |
| --- | --- | --- |
| $this->view  ->assign(‘test’, array(12.36)); |  | {$test[0]} |

Si vous assignez un tableau avec un index alphanumérique, vous pourrez afficher le contenu du tableau.

|  |  |  |
| --- | --- | --- |
| $this->view  ->assign(‘test’, array(‘prenom’ => ‘John’)); |  | {$test.prenom} |

Si vous assignez un tableau avec un index alphanumérique, vous pourrez afficher le contenu du tableau en appelant l’index dynamiquement.

|  |  |  |
| --- | --- | --- |
| $this->view  ->assign(‘test’, array(‘prenom’ => ‘John’))  ->assign(‘index’, ‘prenom’); |  | {$test.$index} |

Si vous assignez un objet, vous pourrez afficher le contenu des paramètres d’objets.

|  |  |  |
| --- | --- | --- |
| $oObject = new \StdClass; $oObject->prenom = ‘john’;  $this->view  ->assign(‘test’, $oObject) |  | {$test->prenom} |

Si vous assignez un objet, vous pourrez afficher le retour d’une de ses méthodes.

|  |  |  |
| --- | --- | --- |
| Class Foo {   public function show() { echo ‘oui’; } }  $oObject = new Foo;  $this->view  ->assign(‘test’, $oObject) |  | {$test->show()} |

Vous pouvez également ensuite mettre des enchaînements des méthodes ci-dessus ainsi que placer des opérations arithmétiques comme {$foo[$x+1]}.

## Les boucles et conditions dans les templates

Pour faire des conditions comme en PHP, il suffit d’écrire de cette façon.

|  |
| --- |
| {if $a == 1}  COOL {elseif $a == 2}  PAS COOL {else}  JE NE SAIS PAS {/if} |

Pour faire des foreach afin de parcourir des éléments, il faudra les écrire de ce façon :

|  |
| --- |
| {foreach from=$aArticles key=$iKey item=$oOneArticle}  {$oOneArticle->get\_title()}<br/> {foreachelse}  PAS D’ARTICLE {/foreach} |

En plus du PHP, les foreach du système de templating permettent de faire un else au cas où l’élément du foreach est vide. Un raccourci très efficace pour des back-office.

Il est également possible d’utiliser une boucle comme un for PHP en utilisant la fonction section du système de templating comme ceci :

|  |
| --- |
| {section start=0 step=1 name=$i loop=20}  {$i}<br/> {/section} |

## Les fonctions dans les templates

Pour vous aider à réaliser des templates très dynamiques, voici une liste de fonctions très utiles existantes.

{append var='test23' value='non'} {$test23[0]}

{append var='test23' value='non' index='monindex'} {$test23.monindex}  
permet de remplir des tableaux numérique et alphanumérique directement dans les templates

{assign var='test24' value='non'} {$test24}  
permet de remplir une variable directement dans un template

## Ajouter des méthodes avec l’ORM dans mon modèle

Les modèles proposent de bases de nombreuses fonctions pour jouer avec les entités que l’on a vu précédemment mais elles peuvent rapidement ne pas suffire au besoin utilisateur. Nous allons donc voir pour créer de nouvelles méthodes en utilisant l’ORM interne afin de créer des modèles nettement plus complexes et qui s’adaptent à nos besoins.

Nous allons rajouter une méthode dans notre modèle article que l’on avait créé par scaffolding. Cependant la table était vide, nous allons donc ajouter des données à l’intérieur :

INSERT INTO `demo`.`article` (`id` ,`title`) VALUES (NULL , 'test1'), (NULL , 'test2');

Ce modèle e:/venus/bundles/src/Demo/app/Model/article.php contient aucune méthode pour le moment. Nous allons ajouter cette méthode ci-dessous en tant que première méthode :

|  |
| --- |
| /\*\*  \* Get my two title  \*  \* @access public  \* @param integer $iLimit limit  \* @param int $iOffset offset  \* @return array  \*/  public function getMyTitles($iLimit = 10, $iOffset = 0) {  $aResult = array();  $aResult['items'] = $this->orm  ->select(array('SQL\_CALC\_FOUND\_ROWS', '\*'))  ->from($this->\_sTableName, 'a')  ->where($this->where->whereEqual('a.title', 'test1')->orWhereEqual('a.title', 'test2'))  ->orderBy(['a.id DESC'])  ->limit($iLimit, $iOffset)  ->load();  $aResult['count'] = $this->orm  ->select(array('FOUND\_ROWS()'))  ->load();  $aResult['pages'] = *floor*($result['count'] / $iLimit);  return $aResult;  } |

Nous avons ici créé notre requête en utilisant un ORM interne complet. Par principe, nous offrons toujours la possibilité de limiter les retours mais également de toujours retourner un tableau avec les items et le count en racine (le page est un complément intéressant quand on veut faire des paginations).

Grâce à cette méthode qui reste très facile à lire, on ajoute le nombre de fonctions nécessaires pour combler des besoins qui vont parfois au-delà des méthodes natives présentées précédemment.

$this->orm est un raccourcis vers une instance de l’ORM afin de simplifier le travail au développeur. Ensuite on appelle différentes méthodes pour créer une requête SQL sous forme d’objet.

Pour faire une requête SELECT avec l’ORM (proche des ORM existants), il faut définir différentes méthodes comme indiquées ci-dessous :

|  |  |
| --- | --- |
| $this->orm->select(array $aSelect) | En mettant SQL\_CALQ\_FOUND\_ROWS en premier, l’ORM comprend automatiquement la demande de préparer un count total. Il faut mettre les champs désirés en tableau. |
| $this->orm->from($this->\_sTableName, ‘a’) | Le $this->\_stableName est la variable qui permet d’inqiquer automatiquement la table du modèle. Il est possible voire conseillé de mettre un deuxième paramètre qui permet d’indiquer un alias. |
| $this->orm->where($oObjectWhere) | Le where doti être complété avec un objet de type Where qui permet de faire des where très détaillé sous forme d’ORM. |
| $this->orm->orderBy($aOrders) | Permet de définir le order by sachant qu’on donne un tableau ce qui permet d’indiquer plusieurs ordres comme en SQL |
| $this->orm->limit($iLimit, $iOffset) | Permet d’inqiquer une limitation et le numéro du premier élément à retourner. Le $iOffset n’est pas obligatoire. |
| $aResult = $this->orm->load(); | Permet de charger la requête préparée précédement et de recevoir le résultat de la base de données. |

Vous pouvez à présent créer vos propres modèles. Vous pouvez vous référer à e:/venus/bundles/lib/Orm.php et e:/venus/bundles/lib/Orm/Where.php pour connaitre l’étendu de l’ORM qui évolue chaque jour.

Nous détaillerons les autres fonctions de l’ORM dans un autre chapitre.

## Rediriger une configuration vers celle d’un autre bundle

Il peut être utile de chercher la configuration d’un autre bundle comme par exemple pour les Base de données. Cela évite de configurer de façon globale à l’application mais également de ne pas dupliquer deux fois le même contenu.

Pour cela vous pouvez ajouter dans votre fichier de configuration Db.conf :

{

"redirect": "Helium"

}

Cela permet d’indiquer le Bundle vers lequel l’application doit se référer.

## Créer des formulaires dynamiques

Venus permet de créer des formulaires dynamiques directement à partir du controller :

$oForm = $this->form

->add('name', 'text')

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\merchant')

->getForm();

On va créer un formulaire en ajoutant un champ texte et un bouton submit. Ensuite on indique l’entité qui doit se synchroniser avec le formulaire (sachant que les champs doivent avoir le nom des colonnes de la table).

Si vous ajoutez un champ caché avec la clé primaire de la table, l’update sera automatique.

**!!!Changement dans Venus 2 1.0.0 !!!**

Le getForm() permet de récupérer un conteneur qui propose différentes méthodes utiles pour afficher le formulaire final ou réaliser d’autres opération.

A/ Ajouter un label

Dans un formulaire, vous pouvez afficher un label personnalisé (et non le name du form) :

->add('name', 'text', 'mon label')

A/ Ajouter une valeur par défaut

Dans un formulaire, vous pouvez ajouter une valeur par défaut :

->add('name', 'text', 'mon label', 'DEFAULT')

Pour proposer un formulaire qui remplit automatiquement celui-ci afin de faire un update, il suffit de rajouter l’id de la clé primaire désirée dans le synchronizeEntity :

$sForm = $this->form

->add('name', 'text')

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\merchant', 12)

->getForm();

Exemple de formulaire avec une liste de selection où $aFinalMerchant est un tableau de type array(id => name) :

$oForm = $this->form

->add('name', 'text', 'Name')

->add('id\_merchant', 'select', 'Merchants', **null**, $aFinalMerchant)

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

Exemple de formulaire avec une liste de champs à cocher directement créés par le framework avec le même tableau :

$oForm = $this->form

->add('name', 'text', 'Name')

->add('id\_merchant', 'select', 'Merchants', $aFinalMerchant)

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

On peut aussi juste ajouter un simple label dans nos formulaires :

Exemple de formulaire avec une liste de selection où $aFinalMerchant est un tableau de type array(id => name) :

$oForm = $this->form

->add('Mon Label', 'label')

->add('id\_merchant', 'select', 'Merchants', **null**, $aFinalMerchant)

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

## Le conteneur retourné par GetForm()

**Disponibilité : Venus2 1.0.0**

Le conteneur propose par exemple d’afficher le formulaire final en une seule variable.

Controlleur : $this->view->assign(‘form’, $oForm->createView());

Vue : {$form->form}

Pour sauvegarder le résultat, il faudra ajouter un $form->handleRequest(). Le synchronizeEntity() n’est pas obligatoire pour valider un formulaire. Par contre si vous faites un synchronizeEntity, le résultat ne s’enregistrera qu’après avoir fait le $form->handleRequest() :

$oForm = $this->form

->add('Mon Label', 'label')

->add('id\_merchant', 'select', 'Merchants', **null**, $aFinalMerchant)

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

$oForm->handleRequest();

If ($oForm->isValid()) { /\*résultats \*/ }

Vous pouvez aussi tester si le formulaire a été soumis (sans en tester la validité) en utilisant la méthode $oForm->isSubmitted().

Vous pouvez tester le bouton submit qui a été cliqué (quand vous en mettez plusieurs) avec la méthode $oForm->isClicked(‘bouton\_valid1’) ;

Dans les templates, vous pouvez écrire tout le formulaire vous-même comme ceci :

{$form->form\_start}  
 {$form->form\_row[‘name’]}   
 {$form->form\_row[‘firstname’]}  
{$form->form\_end}

Pour renforcer la sécurité des formulaires, vous pouvez rajouter des contraintes

## Les formulaire imbriqués

**Disponibilité : Venus2 1.0.1**

Venus 2 permet de faire des formulaires imbriqués comme ceci :

$oForm = $this->form

->add('Utilisateur', 'label')

->add('id\_user', 'select', 'Users', **null**, $aUser)

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

$oFinalForm = new \Venus\lib\Form;

$oForm2 = $oFinalForm

->add('title\_group', 'input', ‘Title Group’)

->add('users', $oForm)

->add('validate', 'submit')

->synchronizeEntity('Venus\src\Helium\Entity\user')

->getForm();

$this->view->assign(‘form’, $oForm2->createView());

Dans les templates, vous pouvez écrire tout le formulaire vous-même comme ceci :

{$form->form\_start}  
 {$form->form\_row[‘users’][‘id\_user’]}   
 {$form->form\_row[‘title\_group’]}  
{$form->form\_end}

## Ajouter des contraintes (validator) sur vos formulaire

**Disponibilité : Venus2 1.0.2**

Voici un exemple d’utilisation de contrainte sur les formulaires ce qui permettra d’invalider ou de valider un formulaire :

$oForm = $this->form

->add('Utilisateur', 'label')

->add('id\_user', 'select', 'Users', **null**, $aUser)

->synchronizeEntity('Venus\src\Helium\Entity\user')

->addConstraint('Utilisateur', new \Venus\lib\Validator\NotBlank)

->getForm();

Le formulaire n’acceptera pas d’avoir un utilisateur vide et retournera une exception.

## Sauvegarde et suppression rapide des entités

Venus permet de sauvegarder ou de modifier rapidement une entité. Pour cela lorsque vous avez une entité faites :

$oEntity->save() ; pour la sauvegarder (update ou insert selon la clé primaire indiquée)  
=> on peut mettre true en paramètre pour accepter un INSERT ON DUPLICATE KEY UPDATE.

$oEntity->remove() ; pour la supprimer

## Gestion des caches

Le framework permet de gérer du cache de différente façon : utiliser des fichiers, utiliser du cache mémoire classique, Redis, Memcache ou Apc.

Pour utiliser un cache, il suffit d’utiliser cette classe :

$oCache = new \Venus\lib\Cache;  
$oCache->setCacheType(‘apc’); // apc, memcache, redis, file, memory

Attention, pour Redis et Memcache vous devez créer des fichiers de configuration globals ou par Portail. Vous trouverez un exemple dans le dossier de configuration global.

## Application multilingue

Venus permet de faire des applications multilingues. Pour cela vous pouvez utiliser la classe I18n qui propose 3 modes : le gettext si vous l’avez activé et que le fichier .mo existe, un traducteur local si le fichier .json existe ou sinon un simple Mock pour vous laisser simuler une traduction.

Une classe Mock permet de ne pas bloquer vos développements en proposant toutes les fonctionnalités des classes réelles mais en ayant aucun résultat réel.

Les fichiers de traductions .mo ou .json doivent se trouver dans /data/i18n/. Un exemple de fichier json est présent dans le framework.

La configuration du multilingue se fait dans le fichiers de configuration Const.json où il faudra configurer « language », « i18n\_domain » et « i18n\_directory ».

Dans un template, vous pourrez mettre {gettext word=’cool’}

## Faire un modèle de données avec un join

Venus permet aux entités de se lier entre elles. Pour cela il suffit de définir un join sur un des champs join dans le Db.conf comme ceci :

"id\_product": {

"type": "int",

"null": false,

"unsigned": true,

"join": "product",

"join\_by\_field": "id"

},

Le join représente la table sur laquelle se lier et join\_by\_field le champs sur lequel faire la jointure. Vous pouvez omettre le deuxième paramètre si le champ a le même nom sur les deux tables.

La jointure se fait automatiquement sur les deux tables donc il est inutile de la déclarer sur les deux tables.

## 3 types d’autoloader possibles dans Venus 2

**Disponibilité : Venus2 1.0.0-beta2**

Tous les autoloader sont gérés dans le fichier /bundles/conf/Autoload.php.

Le premier Autoload est la fonction classique qui intègre l’ensemble du framework Venus2.

Le deuxième autoload intègre le fichier autoloader de composer.json. En cas de besoin, vous pouvez ajouter un package dans /bundles/ext/composer.json et qui sera automatiquement pris en charge par Venus2 après avoir joué composer install ou composer update.

Le troisième autoload intègre les fichiers autoload (ou fichiers simples) définit dans le fichier /bundles/conf/Const.conf comme ceci :

|  |
| --- |
| {  "autoload": [  "bundles/ext/vendor/facebook/base\_facebook.php",  "bundles/ext/vendor/facebook/facebook.php"  ]  } |

## Utiliser un package composer

**Disponibilité : Venus2 1.0.0-beta1**

Venus 2 permet d’ajouter des packages composer pour alimenter le Framework de nouvelles fonctionnalités.

D’ailleurs pour ceux qui préfèrent Twig ou Smarty, ils peuvent rajouter ce gestionnaire de templating par composer. Dans Venus 2, le gestionnaire de templating est devenu indépendant et proposé sous composer sous le nom d’Apollina.

Vous pouvez ajouter votre package dans /bundles/ext/composer.json, relancer la machine en allant dans le dossier /bundles/ext/ et en faisant composer install (ou composer update).

## Utiliser Smarty à la place d’Apollina

**Disponibilité : Venus2 1.0.0-beta1**

Pour utiliser Smarty à la place d’Apollina, vous pouvez ajouter le composer (smarty/smarty) [voir le point précédent] et vous pouvez adapter le fichier /bundles/lib/Vendor.php pour faire fonctionner Smarty à la place d’Apollina.

## 2 triggers d’initialisation sur les entités

**Disponibilité : Venus2 1.0.0-beta2**

Les contrôleurs ont deux triggers qui permettent d’être initialisé :

1/ La fonction initialize permet d’initialiser le contrôleur la première fois qu’il est appelé et uniquement cette fois-là pendant tout le déroulement du script :

|  |
| --- |
| **public function** initialize() { ; } |

2/ La fonction onConstruct permet d’initialiser le contrôleur à chaque fois qu’il est instancié :

|  |
| --- |
| **public function** onConstruct() { ; } |

## Service d’injection

**Disponibilité : Venus2 1.0.0-beta2**

Une grande nouveauté de Venus est de faire des services d’injection avec l’injection de dépendance.

Voici un exemple d’injection de service que l’on peut faire dans un contrôleur :

|  |
| --- |
| $this->di->set(mongo, **function**() {  **return** **new** Mongo();  }, **true**); |

Le true (on peut mettre false) permet de définir si on partage le service à toutes les instances ou uniquement à cette instance. (le true valide le partage pour tous).

Dans un contrôleur, pour l’appeler, il suffit d’utiliser l’une de ces deux méthodes :

|  |
| --- |
| // Directement en appelant mongo, le nom de l’injection  $this->mongo->connect();  // En passant par le raccourcis de l’injection de dépendance  $this->di->get('mongo')->connect(); |

## Objet Request pou récupérer les POST et les PUT

**Disponibilité : Venus2 1.0.0-beta2**

Vous pouvez récupérer le contenu des POST envoyé par formulaire voire tester si il y a un POST envoyé de cette façon :

|  |
| --- |
| **if** ($this->request->isPost() == **true**) {  $sName = $this->request->getPost("name");  $sFirstname = $this->request->getPost("firstname");  } |

Dans un car très particulier comme par exemple lors de la création de web service en Rest, vous pouvez récupérer le contenu du PUT :

|  |
| --- |
| $this->request->getPut(); |

## 2 triggers sur les contrôleurs

**Disponibilité : Venus2 1.0.0-beta2**

Venus 2 vous propose 2 triggers sur les contrôleurs qui se lancent juste avant et juste après l’appel de l’action :

|  |
| --- |
| **public** **function** beforeExecuteRoute()  {  *// s’exécute juste avant l’appel de l’action*  }  **public** **function** afterExecuteRoute()  {  *// s’exécute juste après l’appel de l’action*  } |

## Des clés étrangères

**Disponibilité : Venus2 1.0.0-beta2**

Venus 2 permet de définir des clés étrangères complètes dans les fichiers de configurations de bases de données. Voici un exemple complet :

|  |
| --- |
| "id\_role": {  "type": "int",  "value": "10",  "null": false,  "join": "role",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "role\_fk"  }, |

## Définir le moteur, l’auto-increment et le charset de la table

**Disponibilité : Venus2 1.0.0-beta2**

Venus 2 permet de définir des compléments d’information sur les tables que vous voulez créer comme le moteur utilisé, l’auto-increment de départ et le charset par défaut de la table. Voici un exemple de table avec ce type de définition :

|  |
| --- |
| "board": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "name": {  "type": "varchar",  "value": "50",  "null": false  },  "id\_role": {  "type": "int",  "value": "10",  "null": false,  "join": "role",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "role\_fk"  },  "id\_user": {  "type": "int",  "value": "10",  "null": false,  "join": "user",  "join\_by\_field": "id",  "join\_delete": "cascade",  "join\_update": "cascade",  "constraint": "user\_fk"  }  },  "engine": "InnoDB",  "auto\_increment": 3,  "default\_charset": "utf8"  } |

## Définir des index dans la table

**Disponibilité : Venus2 1.0.0-beta1**

Venus 2 permet de définir des index de deux façons différentes (soit groupé, soit directement sur les champs en question). On ne pourra créer des index multiple que dans la première façon de faire en revanche. Voici deux exemples de table avec ce type de définition :

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true  }  },  "index": [  "id\_team\_name\_index": ["id\_team"]  ],  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

ou

|  |
| --- |
| "user": {  "fields": {  "id": {  "type": "int",  "key": "primary",  "null": false,  "unsigned": true,  "autoincrement": true  },  "id\_team": {  "type": "int",  "null": false,  "unsigned": true,  "key": “index”  }  },  "engine": "InnoDB",  "auto\_increment": 1,  "default\_charset": "utf8"  }, |

## Créer un plugin

**Disponibilité : Venus2 1.0.0-beta2**

Enorme nouveauté dans Venus2, la possibilité de créer des plugins, soit des extensions utilisables au sein du framework.

Vous pourrez donc créer un plugin en ajoutant un dossier dans : /bundles/src/plugins/

Il est également possible de créer un nouveau plugin avec un batch :

/> php bin/console create\_plugin –n MonPlugin

Si mon plugin a besoin d’installer des tables complémentaires, je vais pouvoir l’indiquer dans la méthode spécialement conçue pour installer le plugin :

|  |
| --- |
| /\*\*  \* install method  \*  \* **@access** public  \* **@return** void  \*/  **public function** install()  {  $this->installDb;  } |

Cela va automatiquement lire le fichier Db.conf et va créer les tables, les entités et les modèles correspondants.

Quand on a fini de créer son plugin, il faut installer celui-ci.

En premier lieu, on va rajouter ce plugin dans le fichier /bundles/conf/Plugin.conf

|  |
| --- |
| [  "MonPlugin"  ] |

On voit ensuite lancer le script qui permet d’installer le plugin sur le portail qu’on désire :

\>php bin/console install\_plugin –p MonPortail

## Définit plusieurs host dans une route

**Disponibilité : Venus2 1.0.0**

Afin de faciliter les configurations de host pour avoir les mêmes routes en dev et en prod, on peut maintenant définir plusieurs host dans une route séparés par une virgule :

{

"localhost:85,www.scrum-agile.com" : {

"routes": {

## Gestion d’Asset sous Venus

**Disponibilité : Venus2 1.0.0**

Venus propose pour sa version 2 de gérer les asset. Ce concept permet de gérer des CSS et des JS qui ne sont pas dans le dossier public classique mais dans le dossier public du bundle. Voici un exemple pour ajouter des CSS et des Javascript avec les asset :

|  |
| --- |
| $oAsset = new \Venus\lib\Asset;  $oAsset->stylesheets(array('Demo/public/css/style.css'));  $oAsset->javascripts(array('Demo/public/js/javascript.js')); |

Dans le fichier de configuration Const.conf vous pourrez rajouter une option pour ajouter un paramètre à l’URL des assets ; cela permet de décacher automatiquement les fichiers sur les navigateurs des utilisateurs et sur les cache de type Varnish (Akamai) [on appelle ça du cache busting].

|  |
| --- |
| "asset\_version" : "v2" // ajoute v2 |

Vous pouvez appeler les asset de cette façon :

|  |
| --- |
| <link rel="stylesheet" href="/getCss?Demo/public/css/global"> <link rel="stylesheet" href="/getJs?Demo/public/js/global"> |

ou en utilisant une function templating comme ceci :

|  |
| --- |
| <link rel="stylesheet" href="{asset template="Demo/public/css/global"}"> <link rel="stylesheet" href="{asset template="Demo/public/js/global"}"> |

## Objet Request complet

**Disponibilité : Venus 3.0.0**

Dans cette nouvelle version de Venus Framework, il faudra utiliser un objet Request pour aller chercher un grand nombre d’information comme :

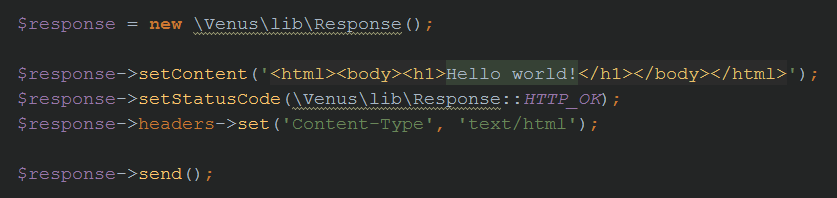
$A = new Request;  
$A->request->get(‘name’) => pour chercher le $\_POST[‘name’];   
$A->cookies->get(‘name’) => pour chercher le $\_COOKIE[‘name’];   
$A->files->get(‘name’) => pour chercher le $\_FILES[‘name’];  
$A->headers->get(‘name’) => pour chercher le header défini par name  
$A->query->get(‘name’) => pour chercher le $\_GET[‘name’];   
$A->server->get(‘name’) => pour chercher le $\_SERVER[‘name’];

Toutes ces méthodes sont utilisées de manières identiques et autorisent également un deuxième paramètre qui est la valeur par défaut retournée si le paramètre n’est pas défini.

## Objet Response complet

**Disponibilité : Venus 3.0.0**

Dans cette nouvelle version de Venus Framework, l’objet Response prend de l’importance.



Ici, on envoie du contenu, on indique un numéro de statut, rajoutons un header HTML avant d’envoyer la totalité au navigateur avec send().

## Les choses qui disparaissent

**Disponibilité : Venus 3.0.0**

De gros changements sont en cours sur cette troisième version du framework :

1/ Le bundle/src/launch.php est transformé par le bin/console  
2/ Les librairies obsolètes Error et Exception sont supprimées

[en cours]